Constructors and Assignment ...

Mike Precup
(mprecup@stanford.edu)
ENJMIN 2016

Why Constructors?

- A constructor is a function which is called when an object is first created
  - Objects are created on the stack by a variable declaration
  - Objects on the heap are created with `new`
- The constructor sets up the initial state of the object for later functions
- This should be familiar, but let's go a bit more in depth...

Why Constructors?

```
class Vector {
    Vector() {
        logicalSize = 0;
        allocatedSize = 8;
        elems = new int[allocatedSize];
    }
};

// Both of these lines call the constructor
Vector x;
Vector *y = new Vector();
```

Why Constructors?

Why do objects have constructors?

Can't we just use an `init` function which does the same thing the constructor does?

Why Constructors?

```
struct foo {
  int value;
  void init(int v) {value = v;}
};

foo x;
x.init(42);
cout << x.value << endl;
```

Why Constructors?

- Issue #1: What if we forgot `x.init()`?

```
struct foo {
  int value;
  void init(int v) {value = v;}
};

foo x;
// x.init(42);
cout << x.value << endl; // What's printed?
```

Why Constructors?

- Issue #2: I'm incredibly lazy.

```
int main() {
    // Construct *and* initialize in one line
    HasAConstructor x(42);
    // .init() requires two lines!
    HasInitFunction y;
    y.init(42);
}
```

Why Constructors?

- Issue #3: Const data members

```
struct ConstMember {
    const int value;
    void init(int v) {value = v;}
};

ConstMember x;
x.init(42); // Error: assignment to const!
```

Why Constructors?

The notion of **initialization** is fundamental to the C++ language and distinct from the notion of **assignment**.

Why Constructors?

Initialization transforms an object's initial junk data into valid data.

Assignment replaces existing valid data with other valid data.

Why Constructors?

Initialization is defined by the **constructor** for a type.

Assignment is defined by the **assignment operator** for a type.

Constructors

We will be looking at three kinds of Constructors today

- Default Constructors
  - What you are used to but with some new tricks
- Copy Constructors
  - Construct an instance of a type to be a copy of another instance
- Copy Assignment
  - Not really a constructor
  - Assign an instance of a type to be a copy of another instance

Why Constructors?

```
// Initialization: Default Constructor

Widget x;

// Initialization: Copy Constructor

Widget y(x);

// Initialization: Copy Constructor (form 2)

Widget z = x;

// Assignment: Copy assignment

z = x;
```

Quick Note

It is not always necessary to define all types of constructors and assignment. If you don’t the compiler will create a default version for you

Constructors

We will be looking at three kinds of Constructors today

- Default Constructors
  - What you are used to but with some new tricks
- Copy Constructors
  - Construct an instance of a type to be a copy of another instance
- Copy Assignment
  - Not really a constructor
  - Assign an instance of a type to be a copy of another instance

Default Constructors

- A constructor looks just like any other member function for a type, with 3 distinctions
  - Constructors have **no return value** listed (not even void)
  - Constructors have the **same name as the type** in question
  - Constructors can have an **initialization list**

Default Constructors

Initialization lists allow us to **initialize** (not assign) data members when we initialize our type.

```
// Assignment
struct Widget {
    const int value;
    Widget();
};
Widget::Widget() {
    value = 42; //ERROR
}
```

```
// Initialization
struct Widget {
    const int value;
    Widget();
};
Widget::Widget()
    : value(42) {}
```

Default Constructors

Initialization lists can have multiple parts

```
struct Person {
    int age;
    string name;
    Person();
};

Person::Person() : age(36), name("Kanye") {
    //Empty constructor since nothing to assign
}
```

Default Constructors

Constructors solve all 3 of the problems with the init function. Value is initialized to v, not assigned

```
struct ConstMember {
    const int value;
    ConstMember(int v) : value(v) {}
};

int main() {
    ConstMember b(42); // value is 42
}
```

Vector Constructors

Let's now take a look at a more complex constructor -- our old friend `Vector<T>`.

Vector Constructors

```
// Initialize an empty vector:
vector<int> a;


// 42 elements: all zero
vector<int> b(42);


// 42 elements: all set to 11
vector<int> c(42, 11);
```

Interlude: Default Parameters

We're about to do something cool, but we need to review default parameters first.

Interlude: Default Parameters

- In C++, we can list **default parameters** for functions which take arguments
- Functions without default parameters can have their rightmost parameters omitted and the default values will be used
- The syntax is simple, but default parameters should only be listed in the *declaration* of a function, not the *definition*

Interlude: Default Parameters

```
// Declare our default arguments
void f(int a, int b = 5, int c = 42);

// Define our function
void f(int a, int b, int c) {
    cout << a << " ";
    cout << b << " ";
    cout << c << endl;
}
```

Interlude: Default Parameters

```
// Declare our default arguments
void f(int a, int b = 5, int c = 42);

// Define our function
void f(int a, int b, int c) {
    cout << a << " ";
    cout << b << " ";
    cout << c << endl;
}

f(1);       // 1 5 42
f(1, 2);    // 1 2 42
f(1, 2, 3); // 1 2 3
```

Constructors

We will be looking at three kinds of Constructors today

- Default Constructors
  - What you are used to but with some new tricks
- Copy Constructors
  - Construct an instance of a type to be a copy of another instance
- Copy Assignment
  - Not really a constructor
  - Assign an instance of a type to be a copy of another instance

Copy Constructors

- A copy constructor is called when an instance of a type is constructed from another instance
- Two ways it can be called

```
//directly call the copy constructor

int x(4);


//implicitly call the copy constructor

int y = 2;
```

Copy Constructors

- A copy constructor is called when an instance of a type is constructed from another instance

```
Vector<string> a(10, “hi”);

//directly call the copy constructor

Vector<string> b(a);


//implicitly call the copy constructor

Vector<string> c = a;
```

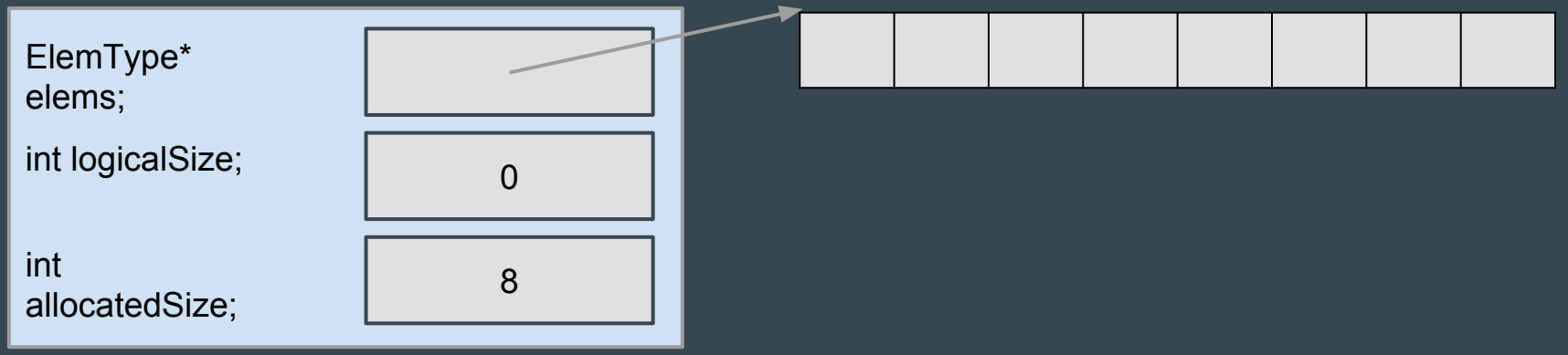
Copy Constructors

Before writing the copy constructor, let's think about how we're going to write it.

- First idea: just copy all of their member variables
- We'll have the correct size and element pointer, so this works right?
- This is what the default copy constructor does

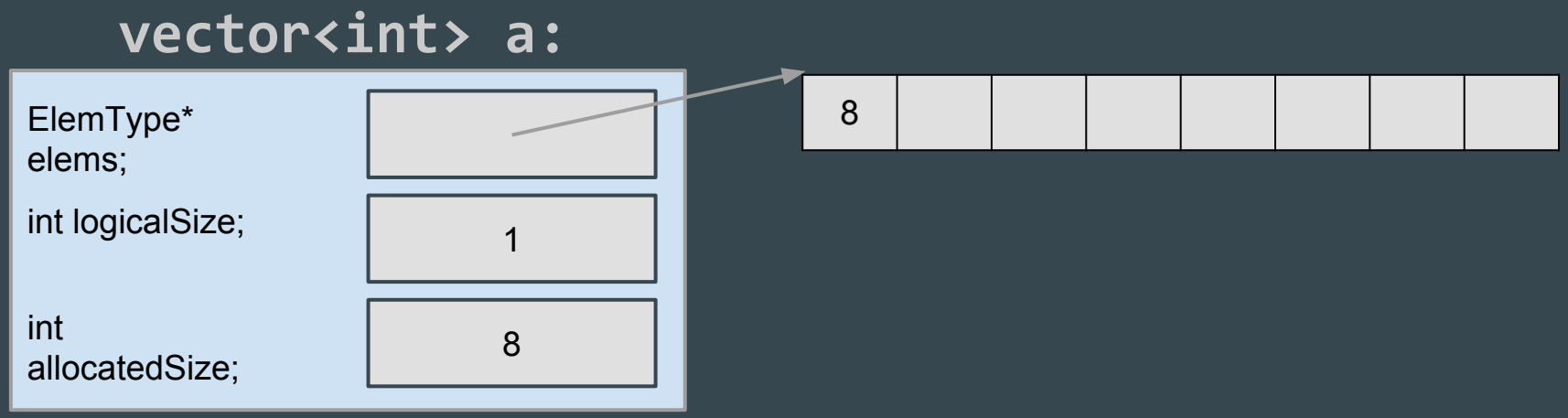
Copy Constructors

`vector<int> a:`



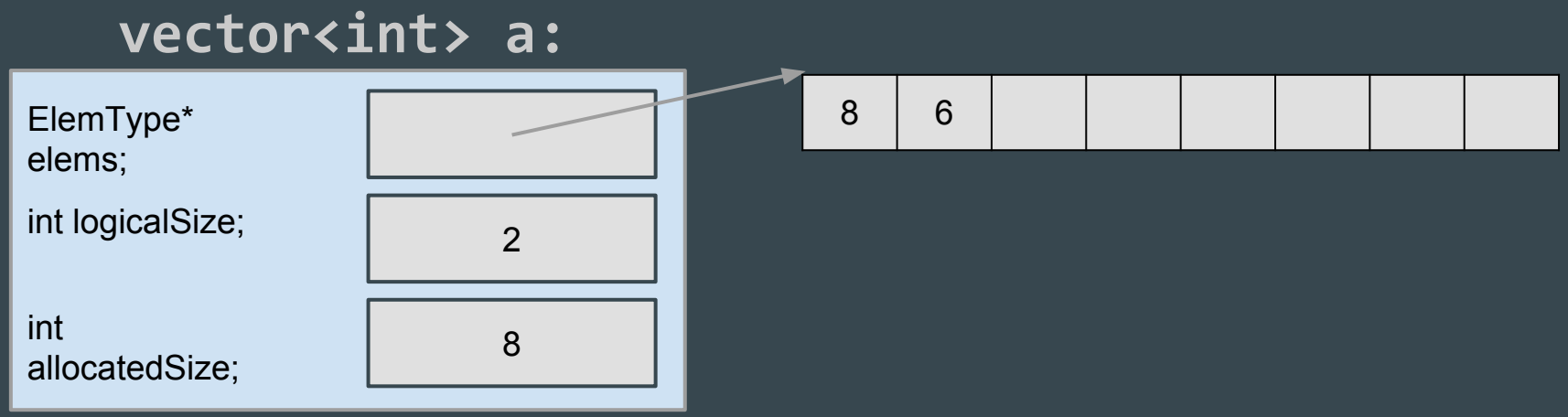

```
Vector<int> a;
```

Copy Constructors



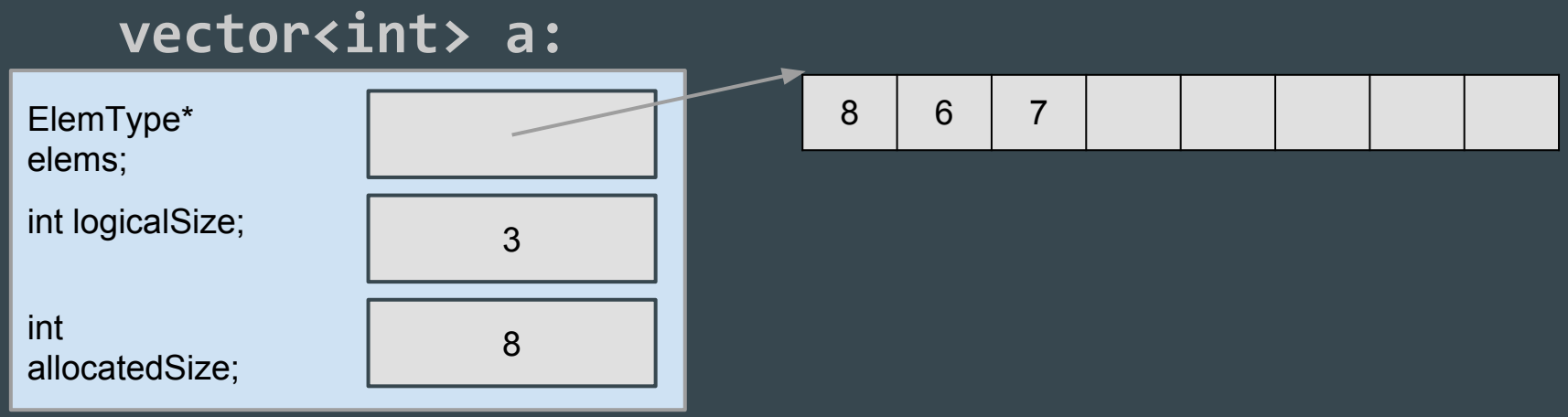

```
Vector<int> a;
a.push_back(8);
```

Copy Constructors



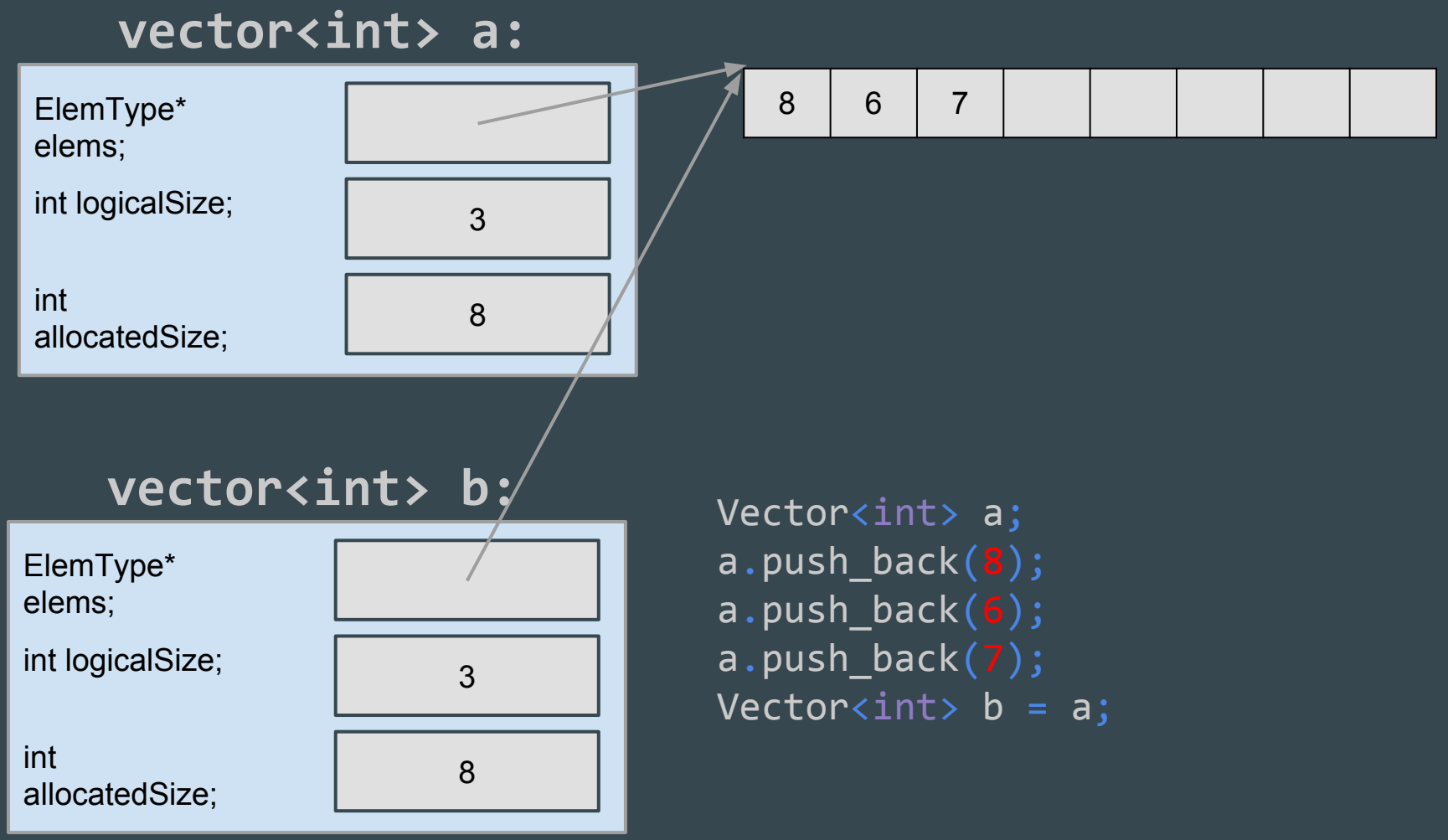

```
Vector<int> a;
a.push_back(8);
a.push_back(6);
```

Copy Constructors



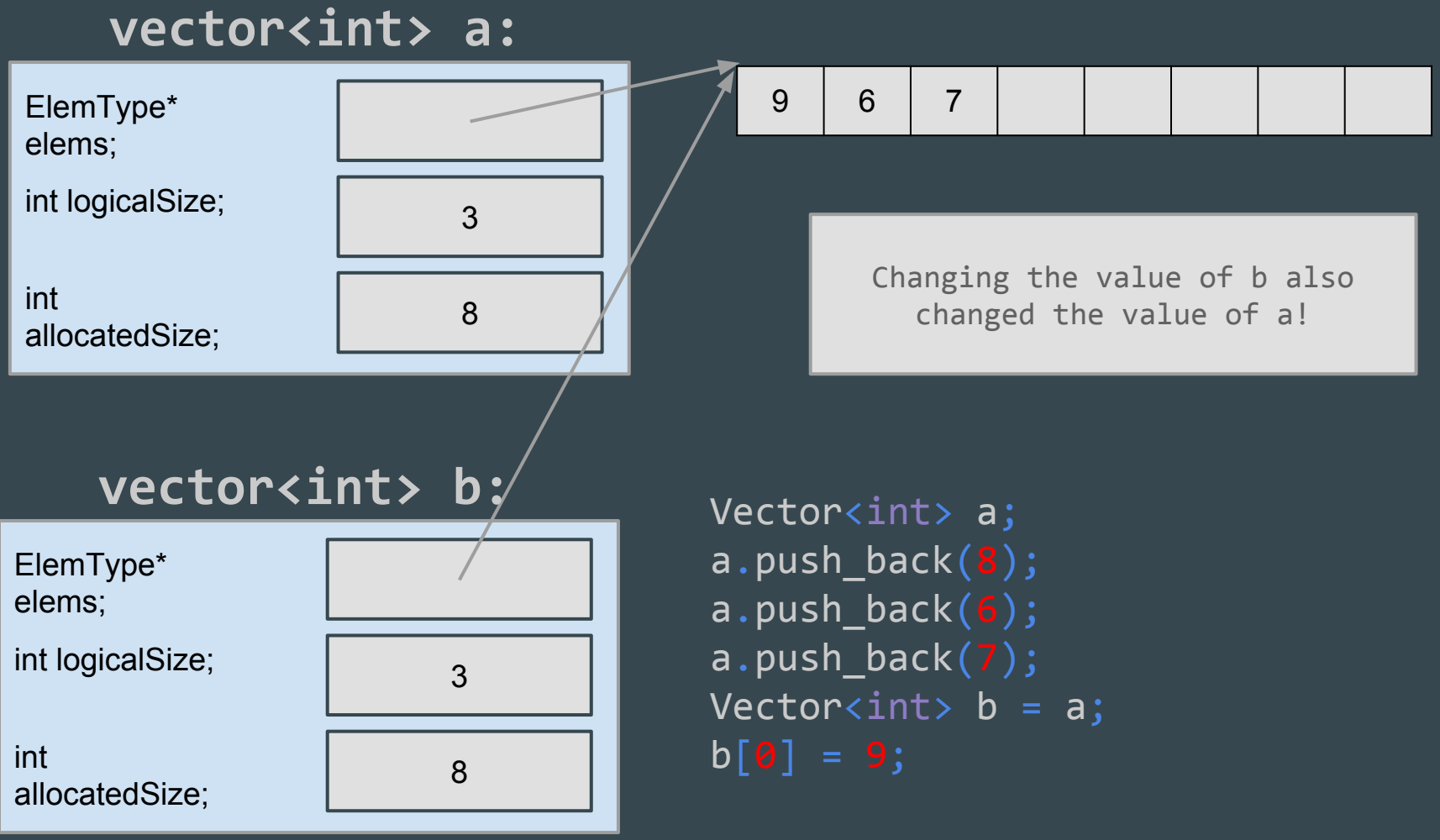

```
Vector<int> a;
a.push_back(8);
a.push_back(6);
a.push_back(7);
```

Copy Constructors



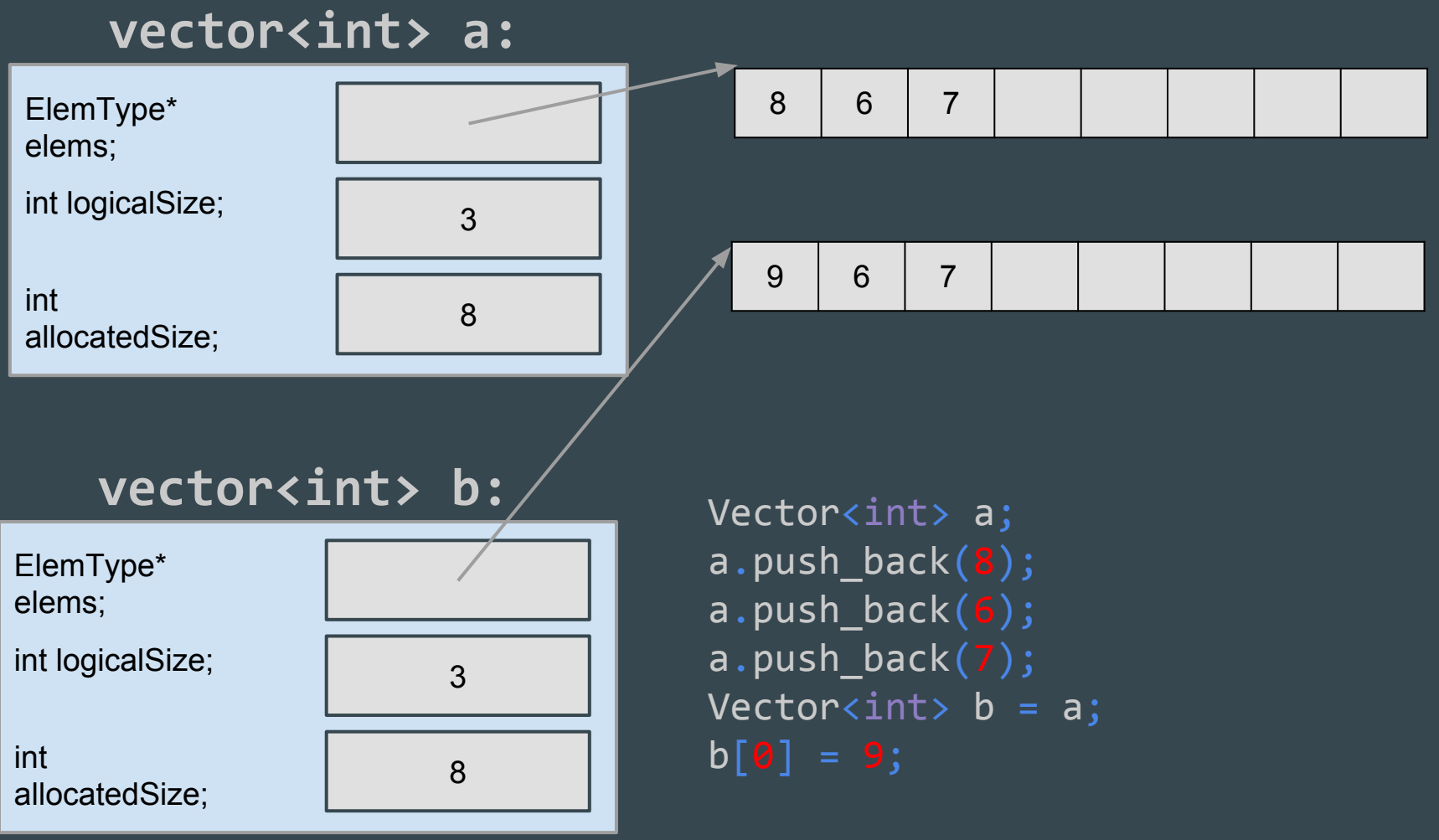

```
Vector<int> a;
a.push_back(8);
a.push_back(6);
a.push_back(7);
Vector<int> b = a;
```

Copy Constructors




```
Vector<int> a;
a.push_back(8);
a.push_back(6);
a.push_back(7);
Vector<int> b = a;
b[0] = 9;
```

Copy Constructors




```
Vector<int> a;
a.push_back(8);
a.push_back(6);
a.push_back(7);
Vector<int> b = a;
b[0] = 9;
```

Constructors

We will be looking at three kinds of Constructors today

- Default Constructors
  - What you are used to but with some new tricks
- Copy Constructors
  - Construct an instance of a type to be a copy of another instance
- Copy Assignment
  - Not really a constructor
  - Assign an instance of a type to be a copy of another instance

Copy Assignment

- Now that we know how to write constructors, include the copy constructor, we're ready to move on to the copy assignment operator
- Remember, assignment takes an already initialized object and gives it new values

```
int x = 4; //Copy Constructor

x = 2; //Copy Assignment
```

Copy Assignment

The syntax for copy assignment is as follows. Note that this is the same syntax as any other operator overload.

```
class Widget {
public:
    Widget& operator=(const Widget& other);
    //Other member vars and functions
};

Widget& Widget::operator=(const Widget& other) {
    // Code to copy data from other
}
```

Copy Assignment

Implementing the **copy assignment operator** is tricky for a couple of reasons:

- Catching memory leaks
- Handling self assignment
- Understanding the return value

Some C++ Quirks

While we are on the topic of constructors, what does this line of code do?

```
Vector<int> v();
```

Some C++ Quirks

While we are on the topic of constructors, what does this line of code do?

```
int x(); //meant to do int x; or int x(2);
```

This is called the **most vexing parse** and has plagued many a C++ programmer. By adding the parentheses, v is treated as a function declaration.

Some C++ Quirks

In our copy constructor the following code will compile

```
Vector<int> x = 15;
```

To prevent accidental type conversions, use the **explicit** keyword before the declaration of your constructor

Some C++ Quirks

A default constructor, copy constructor, and copy assignment operator will all be defined for you if you don’t define them.

Some C++ Quirks

The old way of removing them was to declare them as private, and never implement them. This would create a compiler error if anyone tried to use them.

```
class Vector {

private:

    Vector(const Vector& other);

};

// No implementation
```

Some C++ Quirks

The new way of removing them generates much nicer error messages, and works in all cases. Just set any you don't want equal to `delete`.

```
class Vector {

public:

    Vector(const Vector& other) = delete;

};
```

Rule of Three

If you implement a copy constructor, assignment operator, or destructor, you should implement the others, as well

The Preprocessor

- It's time for a brief note about the **preprocessor**
- The **preprocessor** is the first stage of the compiler
- It does basic text manipulation
- Preprocessor **directives** are prefaced by #

`#include`

The preprocessor isn't terribly intelligent.

When you write `#include`, it's not doing anything fancy,
it just copy pastes the file right in!

#include

Say I have a header file that defines my Vector. If, in main.cpp, I write the following:

```
#include “vector.h”

#include “vector.h”
```

It’s going to define Vector twice!

#include

Now, you might say, eh, I'm smart enough not to write that twice. But sometimes, it's not so obvious:

```
#include “vector.h”

#include “stack.h”
```

If stack uses vector, it's still going to be defined twice!

#ifndef

The typical strategy for solving this problem is to use conditional inclusion of code

These directives allow you to do that:

`#ifndef NAME`

Only include the following code is NAME is not defined (in the preprocessor!)

`#endif`

Close the conditional

`#define NAME`

Define NAME in the preprocessor

#ifndef

The typical structure looks like this:

```
#ifndef L_VECTOR_H_
#define L_VECTOR_H_
// header file code
#endif
```

Make sure you use a name that isn't taken anywhere else

#pragma once

You'll sometimes see this in code, although it's not universally supported

It basically just exactly what you just saw for the entire file

If it's supported and you choose to use it, you can just add that one line to the top!